

Module 15) Advance Python Programming

1. Printing on Screen

1. Introduction to the print() function in Python

The `print()` function in Python is one of the most commonly used **built-in functions**. It is used to display output on the screen, such as text, numbers, variables, or results of expressions.

1. Basic Syntax:

```
print(object, sep=' ', end='\n')
```

- `object` → The value(s) you want to print.
- `sep` → Separator between multiple values (default is a space ' ').
- `end` → Determines what is printed at the end (default is a newline '\n').

Example 1: Printing Text

```
print("Hello, Python!")
```

Output:

Hello, Python!

Example 2: Printing Multiple Values

```
print("My name is", "Rahul", "and I am", 20)
```

Output:

My name is Rahul and I am 20

Example 3: Using sep and end Parameters

```
print("Python", "is", "fun", sep="-", end="!")
```

Output:

Python-is-fun!

Example 4: Printing Variables

```
name = "Bhautik"  
age = 21  
print("Name:", name, "Age:", age)
```

2.Formatting outputs using f-strings and format()

In Python, output formatting is used to display information in a clear, readable, and structured way.

Two common methods for formatting strings are f-strings and the format() method.

1. Using f-Strings (Formatted String Literals)

Introduced in Python 3.6, f-strings allow you to embed variables directly inside string literals using curly braces {}.

They are fast, simple, and readable.

Syntax:

```
f"Your text {variable}"
```

Example:

```
name = "Bhautik"  
age = 21  
print(f"My name is {name} and I am {age} years old.")
```

Output:

My name is Bhautik and I am 21 years old.

You can also perform expressions inside braces:

```
print(f"Next year, I will be {age + 1} years old.")
```

Output:

Next year, I will be 22 years old.

2. Using the format() Method

The format() method inserts values into placeholders {} within a string.

Syntax:

```
"string {} {}".format(value1, value2)
```

Example:

```
name = "Rahul"  
age = 20  
  
print("My name is {} and I am {} years old.".format(name, age))
```

Output:

My name is Rahul and I am 20 years old.

You can also use index numbers or named placeholders:

```
print("My name is {0} and I am {1} years old.".format("Bhautik", 21))  
  
print("My name is {n} and I am {a} years old.".format(n="Rahul", a=20))
```

2. Reading Data from Keyboard

1 Using the `input()` function to read user input from the keyboard.

In Python, the `input()` function is used to **take input from the user** through the **keyboard**. It always returns the user's input as a string, which can then be converted to other data types (like integer or float) if needed.

1. Basic Syntax:

```
variable = input("Enter your message: ")
```

- The text inside quotes is called a **prompt message** — it tells the user what to enter.
- The value entered by the user is stored in the **variable**.

Example 1: Reading a String Input

```
name = input("Enter your name: ")  
  
print("Hello, ", name)
```

Output:

Enter your name: Bhautik

Hello, Bhautik

Example 2: Reading Numeric Input

Since `input()` returns data as a string, you must **convert it** to a number using `int()` or `float()`.

```
age = int(input("Enter your age: "))
```

```
print("You are", age, "years old.")
```

Output:

Enter your age: 21

You are 21 years old.

Example 3: Using Multiple Inputs

```
a = int(input("Enter first number: "))

b = int(input("Enter second number: "))

print("Sum =", a + b)
```

2. Converting user input into different data types (e.g., int, float, etc.).

In Python, the **input()** function always takes user input as a **string**, even if the user types a number.

To perform mathematical operations or comparisons, we often need to **convert** this input into other data types such as integer (int), float, etc.

1. Using Type Conversion Functions

Python provides built-in functions to convert data types:

Function	Description	Example
int()	Converts input to integer	int("10") → 10
float()	Converts input to floating-point number	float("12.5") → 12.5
str()	Converts input to string	str(10) → "10"
bool()	Converts input to boolean (True/False)	bool(1) → True

Example 1: Converting Input to Integer

```
age = int(input("Enter your age: "))

print("Next year you will be", age + 1)
```

Output:

Enter your age: 20

Next year you will be 21

Example 2: Converting Input to Float

```
price = float(input("Enter the price: "))

print("Total price with tax:", price + (price * 0.05))
```

Example 3: String Conversion

```
num = 10

text = str(num)

print("The number is " + text)
```

3. Opening and Closing Files

1. Opening files in different modes ('r', 'w', 'a', 'r+', 'w+').

In Python, files are opened using the **open()** function, which allows reading and writing data from and to files.

The **mode** in which a file is opened determines the kind of operations that can be performed — such as **reading**, **writing**, or **appending**.

Syntax:

```
file = open("filename.txt", "mode")
```

1. Read Mode – 'r'

- Opens a file for reading only.
- The file must already exist, otherwise it will cause an error.
- File pointer is placed at the beginning of the file.

Example:

```
file = open("data.txt", "r")

content = file.read()

print(content)

file.close()
```

2. Write Mode – 'w'

- Opens a file for writing only.
- If the file exists, it is overwritten.
- If the file does not exist, it is created.

Example:

```
file = open("data.txt", "w")
file.write("Hello, Python!")
file.close()
```

3. Append Mode – 'a'

- Opens a file for appending (adding data at the end).
- Existing data is not erased.
- If the file doesn't exist, it is created.

Example:

```
file = open("data.txt", "a")
file.write("\nNew line added.")
file.close()
```

4. Read and Write Mode – 'r+'

- Opens a file for both reading and writing.
- The file must exist.
- The file pointer starts at the beginning.

Example:

```
file = open("data.txt", "r+")
print(file.read())
file.write("\nExtra data.")
file.close()
```

5. Write and Read Mode – 'w+'

- Opens a file for reading and writing.
- If the file exists, it is **overwritten**; if not, it is created.

Example:

```
file = open("data.txt", "w+")
file.write("New content.")
file.seek(0)
print(file.read())
file.close()
```

2) Using the open() function to create and access files.

In Python, the `open()` function is used to create, read, or write files. It allows a program to interact with files stored on the computer.

Syntax:

```
file = open("filename", "mode")
```

- filename – name of the file to open or create
- mode – defines the purpose (read, write, append, etc.)

Common Modes:

Mode	Description
'r'	Opens a file for reading (file must exist).
'w'	Opens a file for writing (creates a new file or overwrites existing one).
'a'	Opens a file for appending (adds data at the end).
'r+'	Opens a file for reading and writing.
'w+'	Opens a file for reading and writing (overwrites existing file).

Example 1: Creating and Writing to a File

```
file = open("example.txt", "w")
file.write("Hello, this is a test file.")
file.close()
```

Example 2: Reading a File

```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

Example 3: Appending to a File

```
file = open("example.txt", "a")
file.write("\nAdding new content.")
file.close()
```

3) Closing files using close()

In Python, after performing file operations like reading, writing, or appending, it is important to close the file using the `close()` method.

This ensures that all data is properly saved and system resources used by the file are released.

Syntax:

```
file.close()
```

Why Closing a File Is Important:

1. It ensures that all changes (write or append) are saved correctly.
2. It frees up system resources and avoids memory leaks.
3. It prevents data corruption or unexpected errors in future file operations.

Example:

```
file = open("example.txt", "w")
file.write("Hello, this is a test.")
file.close()
```

Alternative Method – Using with Statement:

Python also provides an easier way to handle files using the **with** statement. Files opened using with are automatically closed after the block is executed.

```
with open("example.txt", "r") as file:
```

```
    data = file.read()
```

```
    print(data)
```

4. Reading and Writing Files

1) Reading from a file using read(), readline(), readlines().

In Python, after opening a file in read mode ('r') using the open() function, you can read its contents using different methods such as read(), readline(), and readlines().

Each method reads the file in a different way depending on your requirement.

1. read() Method

- Reads the entire content of the file as a single string.
- You can also specify the number of characters to read.

Example:

```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

2. readline() Method

- Reads one line at a time from the file.
- Each time it is called, it reads the next line.

Example:

```
file = open("example.txt", "r")
line1 = file.readline()
line2 = file.readline()
print(line1)
print(line2)
file.close()
```

3. readlines() Method

- Reads all lines of the file and returns them as a list, where each line is a list element.

Example:

```
file = open("example.txt", "r")
lines = file.readlines()
print(lines)
file.close()
```

2) Writing to a file using write() and writelines()

In Python, you can write data to a file using the write() and writelines() methods.

Before writing, the file must be opened in write mode ('w'), append mode ('a'), or read and write mode ('w+' or 'r+') using the open() function.

1. write() Method

- The write() method writes a single string to the file.
- If the file already exists, it overwrites the previous content.
- You can include \n to move to a new line.

Example:

```
file = open("example.txt", "w")
file.write("Hello, Python!\n")
file.write("Welcome to file handling.")
file.close()
```

2. writelines() Method

- The writelines() method writes a list of strings to the file.
- It does not automatically add new lines, so you must include \n manually.

Example:

```
file = open("example.txt", "w")
lines = ["Python is fun.\n", "File handling is easy.\n", "Practice makes perfect.\n"]
file.writelines(lines)
file.close()
```

5. Exception Handling

1) Introduction to exceptions and how to handle them using try, except, and finally.

An exception in Python is an error that occurs during the execution of a program. When an exception happens, the normal flow of the program stops unless the error is handled.

Common exceptions include `ZeroDivisionError`, `TypeError`, `ValueError`, etc.

Exception handling ensures that the program does not crash and can continue running smoothly.

Handling Exceptions Using try, except, and finally

Python provides a structured way to handle exceptions using three main keywords:

1. try Block

- The `try` block contains the code that may cause an exception.
- If no error occurs, the `except` block is skipped.

Example:

```
try:
```

```
    num = 10 / 0
```

2. except Block

- The `except` block runs when an error occurs in the `try` block.
- It is used to handle the exception and prevent program termination.

Example:

```
try:
```

```
    num = 10 / 0
```

```
except ZeroDivisionError:
```

```
    print("You cannot divide by zero.")
```

3. finally Block

- The `finally` block always executes whether an exception occurs or not.
- Commonly used to close files or release resources.

Example:

```
try:  
    file = open("data.txt")  
except FileNotFoundError:  
    print("File not found.")  
finally:  
    print("Execution completed.")
```

Complete Example:

```
try:  
    a = int(input("Enter a number: "))  
    b = int(input("Enter another number: "))  
    result = a / b  
    print("Result:", result)  
except ZeroDivisionError:  
    print("Error: Division by zero is not allowed.")  
except ValueError:  
    print("Error: Invalid input. Please enter numbers only.")  
finally:  
    print("Program Finished.")
```

2) Understanding multiple exceptions and custom exceptions.

Sometimes a block of code may cause different types of errors, and each error needs to be handled differently.

Python allows handling multiple exceptions using multiple except blocks.

Handling Multiple Exceptions Separately

```
try:  
    a = int("abc")      # ValueError  
    b = 10 / 0         # ZeroDivisionError  
except ValueError:  
    print("Invalid input! Cannot convert to integer.")
```

```
except ZeroDivisionError:  
    print("Division by zero is not allowed.")
```

Each except block handles a specific error type.

Handling Multiple Exceptions in One Block

If you want one message for multiple error types:

```
try:  
    x = int(input("Enter number: "))  
    y = 10 / x  
  
except (ValueError, ZeroDivisionError):  
    print("Either input was wrong or division by zero occurred.")
```

Using a Generic Exception

Catches *any* exception:

```
try:  
    a = 10 / 0  
  
except Exception as e:  
    print("An error occurred:", e)
```

This is useful for debugging but should be used carefully.

2. Custom Exceptions

Python also allows you to create your own exception types.

Custom exceptions are useful when you want to handle errors based on your own program logic, not just built-in errors.

Creating a Custom Exception

Create a class that inherits from Exception:

```
class AgeError(Exception):  
    pass
```

Using Custom Exception

```
class AgeError(Exception):  
    pass
```

```
age = 15

try:
    if age < 18:
        raise AgeError("Age must be 18 or above.")

    else:
        print("Eligible.")

except AgeError as e:
    print("Error:", e)
```

Why Use Custom Exceptions?

- Helps create meaningful error messages
- Makes your program easier to debug
- Useful for business logic, e.g., banking, school systems, login systems

6. Class and Object (OOP Concepts)

1) Understanding the concepts of classes, objects, attributes, and methods in Python

1. Classes

A class in Python is a blueprint or template used to create objects.

It defines how an object should look and behave.

A class contains attributes (variables) and methods (functions).

Example:

```
class Car:
```

```
    color = "Red"

    def start(self):
        print("Car started")
```

2. Objects

An object is an instance of a class.

When you create an object from a class, you get a real working version of that class.

Example:

```
my_car = Car() # object creation
```

Here, `my_car` is an object of the class `Car`.

3. Attributes

Attributes are variables inside a class that describe the properties of an object.

There are two types:

a) Class Attributes

- Shared by all objects of the class
- Defined inside the class

Example:

`class Car:`

`wheels = 4`

b) Instance Attributes

- Unique for each object
- Defined inside the constructor (`__init__`)

Example:

`class Car:`

```
def __init__(self, brand, color):  
    self.brand = brand  
    self.color = color
```

4. Methods

Methods are functions defined inside a class.

They define the behavior of an object (what it can do).

Example:

`class Car:`

```
def start(self):  
    print("Car is starting")
```

Here, `start()` is a method.

2) Difference between local and global variables.

Feature	Local Variable	Global Variable
Where it is defined?	Inside a function	Outside all functions (at the top level)
Scope (Where it can be used?)	Only inside the function where it is defined	Can be used anywhere in the program (inside or outside functions)
Lifetime	Exists only until the function finishes	Exists until the entire program ends
Access inside a function	Directly accessible	Must be declared using global keyword if you want to modify it
Access outside a function	Not accessible	Accessible everywhere

7. Inheritance

1) Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.

A child class inherits from one parent class.

Example

class A:

```
def showA(self):
    print("This is class A")
```

class B(A): # Single inheritance

```
def showB(self):
    print("This is class B")
```

obj = B()

obj.showA()

obj.showB()

2. Multilevel Inheritance

Inheritance occurs in multiple levels (grandparent → parent → child).

Example

class A:

```
def showA(self):  
    print("Class A")
```

class B(A):

```
def showB(self):  
    print("Class B")
```

class C(B): # Multilevel

```
def showC(self):  
    print("Class C")
```

```
obj = C()
```

```
obj.showA()
```

```
obj.showB()
```

```
obj.showC()
```

3. Multiple Inheritance

A child class inherits from two or more parent classes.

Example

class A:

```
def showA(self):  
    print("Class A")
```

class B:

```
def showB(self):  
    print("Class B")
```

```
class C(A, B): # Multiple inheritance  
    def showC(self):  
        print("Class C")
```

```
obj = C()  
obj.showA()  
obj.showB()  
obj.showC()
```

4. Hierarchical Inheritance

One parent class, but multiple child classes.

Example

class A:

```
def showA(self):  
    print("Class A")
```

class B(A): # Child 1

```
def showB(self):  
    print("Class B")
```

class C(A): # Child 2

```
def showC(self):  
    print("Class C")
```

```
obj1 = B()
```

```
obj2 = C()
```

```
obj1.showA()
```

```
obj2.showA()
```

5. Hybrid Inheritance

A combination of two or more types of inheritance (e.g., multiple + hierarchical).

Example

A

/ \

B C

\ /

D

class A:

```
def showA(self):  
    print("Class A")
```

class B(A):

```
def showB(self):  
    print("Class B")
```

class C(A):

```
def showC(self):  
    print("Class C")
```

class D(B, C): # Hybrid (Multiple + Hierarchical)

```
def showD(self):  
    print("Class D")
```

obj = D()

obj.showA()

obj.showB()

obj.showC()

```
obj.showD()
```

2) Using the super() function to access properties of the parent class.

Using super() to Access Parent Class Properties

super() is used in Python to access parent class methods, attributes, or constructors from the child class.

It is mostly used in:

- Method Overriding
- Calling Parent Constructor (`__init__`)
- Multiple / Multilevel inheritance

Example 1: Using super() to Call Parent Class Method

class Parent:

```
def show(self):  
    print("This is Parent class")
```

class Child(Parent):

```
def show(self):  
    super().show()    # calling parent method  
    print("This is Child class")
```

```
obj = Child()
```

```
obj.show()
```

✓ Output

This is Parent class

This is Child class

Example 2: Using super() to Call Parent Constructor

class Parent:

```
def __init__(self):  
    print("Parent constructor")
```

```
class Child(Parent):  
    def __init__(self):  
        super().__init__() # calling parent __init__  
        print("Child constructor")
```

obj = Child()

✓ Output

Parent constructor

Child constructor

Example 3: Using super() in Multilevel Inheritance

class A:

```
def show(self):  
    print("Class A")
```

class B(A):

```
def show(self):  
    super().show()  
    print("Class B")
```

class C(B):

```
def show(self):  
    super().show()  
    print("Class C")
```

obj = C()

obj.show()

✓ Output

Class A

Class B

Class C

Why use super()?

- Helps reuse parent class code.
- Avoids repeating the same methods in child classes.
- Useful in multiple inheritance (Method Resolution Order - MRO).
- Makes code clean and maintainable.

8. Method Overloading and Overriding

1) Method overloading: defining multiple methods with the same name but different parameters.

Method overloading means having multiple methods with the same name but different parameters.

Python does not support traditional method overloading like Java or C++.

If you define multiple methods with the same name, the latest method overrides the previous ones.

However, Python allows us to simulate method overloading using:

1. default parameters
2. variable-length arguments (*args)

Example using default parameters:

class Example:

```
def show(self, a=None, b=None):  
    if a is not None and b is not None:  
        print(a + b)  
    elif a is not None:  
        print(a)  
    else:  
        print("No values")
```

```
obj = Example()  
obj.show()  
obj.show(10)  
obj.show(10, 20)
```

2) Method overriding: redefining a parent class method in the child class.

Method overriding happens when a child class defines a function with the same name as in the parent class, replacing the parent's version.

This allows the child class to provide its own behavior.

Example:

class Parent:

```
def show(self):  
    print("This is the parent class")
```

class Child(Parent):

```
def show(self):  
    print("This is the child class")
```

```
obj = Child()
```

```
obj.show()
```

9. SQLite3 and PyMySQL (Database Connectors)

1) Introduction to SQLite3 and PyMySQL for database connectivity

SQLite3 is a built-in Python module used to connect to SQLite databases.

It does not require any installation because it comes with Python.

Key points:

- Lightweight and file-based database.
- No server needed.
- Good for small and medium applications.

Basic steps to use SQLite3:

1. Import the module
2. Connect to a database file
3. Create a cursor
4. Execute SQL queries
5. Commit changes
6. Close the connection

Example:

```
import sqlite3
```

```
con = sqlite3.connect("student.db")
cur = con.cursor()

cur.execute("CREATE TABLE IF NOT EXISTS students(id INTEGER, name TEXT)")
con.commit()
con.close()
```

PyMySQL – Introduction

PyMySQL is a Python module used to connect to a MySQL database.

Unlike SQLite, MySQL uses a database server, so you must install PyMySQL.

Install:

```
pip install pymysql
```

Key points:

- Used for larger, multi-user applications.
- Requires MySQL server running.
- Supports remote connections.

Basic usage:

```
import pymysql
```

```
con = pymysql.connect(
```

```
host="localhost",
user="root",
password="",
database="college"

)
```

```
cur = con.cursor()
cur.execute("SELECT * FROM students")
```

```
for row in cur:
```

```
    print(row)
```

```
con.close()
```

2) Creating and executing SQL queries from Python using these connectors.

1. Executing SQL Queries Using SQLite3

Step-1: Import and connect

```
import sqlite3
```

```
con = sqlite3.connect("student.db")
```

```
cur = con.cursor()
```

Step-2: Create a table

```
cur.execute("CREATE TABLE IF NOT EXISTS students(id INTEGER, name TEXT)")
```

Step-3: Insert data

```
cur.execute("INSERT INTO students VALUES(1, 'Rahul')")
```

```
con.commit()
```

Step-4: Read data (Select query)

```
cur.execute("SELECT * FROM students")
```

```
rows = cur.fetchall()
```

```
print(rows)
```

Step-5: Update data

```
cur.execute("UPDATE students SET name='Amit' WHERE id=1")  
con.commit()
```

Step-6: Delete data

```
cur.execute("DELETE FROM students WHERE id=1")  
con.commit()
```

Step-7: Close connection

```
con.close()
```

2. Executing SQL Queries Using PyMySQL

Step-1: Import and connect

```
import pymysql
```

```
con = pymysql.connect(
```

```
    host="localhost",  
    user="root",  
    password="",  
    database="college"
```

```
)
```

```
cur = con.cursor()
```

Step-2: Create a table

```
cur.execute("CREATE TABLE IF NOT EXISTS students(id INT, name VARCHAR(50))")
```

Step-3: Insert data

```
cur.execute("INSERT INTO students VALUES(1, 'Rahul')")  
con.commit()
```

Step-4: Read data

```
cur.execute("SELECT * FROM students")  
rows = cur.fetchall()  
print(rows)
```

Step-5: Update data

```
cur.execute("UPDATE students SET name='Amit' WHERE id=1")  
con.commit()
```

Step-6: Delete data

```
cur.execute("DELETE FROM students WHERE id=1")  
con.commit()
```

Step-7: Close connection

```
con.close()
```

10. Search and Match Functions

1) Using `re.search()` and `re.match()` functions in Python's `re` module for pattern matching

Python's `re` module is used for pattern matching using regular expressions.

You must import the module first:

```
import re
```

```
re.match()
```

`re.match()` checks only at the beginning of the string.

If the pattern is found at the start, it returns a match object.

If not, it returns `None`.

Example:

```
import re
```

```
text = "Hello Python"
```

```
result = re.match("Hello", text)
```

```
print(result)
```

This matches because "Hello" is at the start.

Example (no match):

```
result = re.match("Python", text)
```

```
print(result)
```

This returns None because "Python" is not at the start.

```
re.search()
```

re.search() checks the entire string for the first occurrence of the pattern.

It returns a match object if found anywhere.

Example:

```
import re
```

```
text = "Hello Python"
```

```
result = re.search("Python", text)
```

```
print(result)
```

2) Difference between search and match.

```
re.match()
```

- Checks for a pattern only at the beginning of the string.
- Returns a match only if the string starts with the pattern.

Example:

```
re.match("Hello", "Hello World") # Match
```

```
re.match("World", "Hello World") # No match
```

```
re.search()
```

- Checks the pattern anywhere in the entire string.
- Returns a match if the pattern appears at any position.

Example:

```
re.search("World", "Hello World") # Match
```

